**Algorithms and Data Structures (ADS) - COMP1819**

Develop and optimise solutions in Python with ADS and provide complexity analysis.

Group Name: 10-02

Team members:

|  |  |  |  |
| --- | --- | --- | --- |
| Member | Name | ID | Contribution % |
| 1 | Raut, Nihaar | 001309432 | 100% |
| 2 | Md. Nellikkaparambu, Md. Shuhaib | 001294977 | 100% |
| 3 | Lemoudden, Ilyass | 001355478 | 100% |
| 4 | Ali, Wais | 001324310 | 100% |
| 5 | Hernandez, Zak | 001355486 | 100% |
|  |  |  |  |

Contents

[1. Create unique solutions! 3](#_Toc158028396)

[**Student 1-6’s solution** 3](#_Toc158028397)

[**Results** 3](#_Toc158028398)

[2. Test and analyse your solution! 4](#_Toc158028399)

[**Your test cases:** 4](#_Toc158028400)

[**Running time graphs** 4](#_Toc158028401)

[**Complexity analysis** 4](#_Toc158028402)

[3. Optimise solutions! 4](#_Toc158028403)

[**Solution 1-2:** 5](#_Toc158028404)

[**Results** 5](#_Toc158028405)

[4. Compare the performance! 5](#_Toc158028406)

[**Time complexities and big-O notations** 6](#_Toc158028407)

[**Running time graphs** 6](#_Toc158028408)

[5. Reflecting on teamwork! 6](#_Toc158028409)

[**Contribution mark** 6](#_Toc158028410)

[**Limitation discussion** 7](#_Toc158028411)

[**Weekly journal** 7](#_Toc158028412)

[Reference 7](#_Toc158028413)

[Appendix A.1 - Proposed solution 1 - 6 8](#_Toc158028414)

[Appendix B - Test cases for correctness 8](#_Toc158028415)

[Appendix C - Evidence of team contribution 9](#_Toc158028416)

# Create unique solutions!

### **Student 1: Raut, Nihaar**

**Understanding of the Problem and approach to solving it**

The journey to find an effecting way to tackle this problem began with going on a comprehensive research endeavour lasting one week to delve into the intricacies of palindromic and prime number generation. Recognising the inefficiency of conventional string-based palindrome checks, it was key to find alternative methods to efficiently generate palindromic numbers.

Through extensive research and discussions with peers, it became evident that generating primes most efficiently would be crucial for handling large test cases, even reaching into the trillions. This realisation led to prioritize optimizing prime number generation techniques over traditional is\_prime functions.

Furthermore, exploration in the development of a custom palindrome generation approach felt more important. This approach aimed to minimise the sample space necessary for the prime function merger to work with, reducing the number of palindromes to be checked for special numbers.

The main idea and approach involved maximising the number of generated primes while simultaneously cutting as much sample space as possible for palindrome generation. This strategy laid the foundation for creating a functional tool capable of efficiently capturing and printing special numbers.

**Description of code 1 (Original Solution)**

**Seiver:** This approach uses a segmented sieve of Eratosthenes for the prime number generation method. It involves breaking down the sieve algorithm into smaller segments to efficiently generate primes within a given range. The segmented sieve optimises memory usage by sieving smaller segments separately, reducing the overall memory footprint required for prime number generation. This segmentation allows for the generation of large prime numbers while maintaining computational efficiency.

The choice of the segmented sieve of Eratosthenes was made based on the understanding that traditional sieving methods and is\_prime might encounter memory limitations and time constraints when handling extremely large ranges, such as those reaching into the trillions.

By segmenting the sieve, the individual ensured that only a portion of the entire range needed to be sieved at any given time, thereby mitigating memory constraints, and improving overall performance.

A grid of numbers with black text

Description automatically generated with medium confidence

**Palindrome Machine:** The palindrome machine is specially crafted to efficiently generate palindromic numbers within a given range. It operates by iteratively constructing palindromes of increasing length, taking advantage of the symmetry inherent in palindromic numbers.

To achieve this, the palindrome machine first initialises a lower and higher digit bound, with the lower bound starting at 1. It then enters an infinite loop, where it loops through each length of palindrome from the length of the lower bound to the length of the upper bound.

For each length, the machine generates palindromes by concatenating the first half of the palindrome with its reverse. If the length of the palindrome is odd, it also loops through all possible middle digits which can be added in to create palindromes with odd lengths.

This approach ensures that the palindrome machine efficiently covers all possible palindromes within the specified range, incrementally increasing the length of the palindrome with each iteration. By generating palindromes in this manner, the machine minimises unnecessary memory usage and computational overhead, resulting in a streamlined and efficient process for generating palindromic numbers.

**Merger:** The merger function is responsible for combining the generated palindromes with prime numbers to identify special numbers within a given range. It utilises the segmented sieve to efficiently generate prime numbers within the square root of the upper bound of the specified range.

The segmented sieve then divides the range into smaller segments, each of which is individually sieved for prime numbers. This segmented approach allows for better memory management and performance, particularly for larger ranges.

Once the palindromes and prime numbers are generated, the merger function iterates through each palindrome, checking if it is also a prime number. If a palindrome is prime, it is added to the list of special numbers.

By accumulating the segmented sieve of Eratosthenes and efficiently combining palindromes with prime numbers, the merger function effectively identifies special numbers within the specified range while minimizing computational overhead.

**Difference in Code [Highlighted Section]**

1. **class** SpecialNumbers:
2. @staticmethod
3. **def** prime\_siever(limit):
4. primes **=** []
5. sieve **=** [True] **\*** (limit **+** 1)
6. sieve[0] **=** sieve[1] **=** False
7. **for** p **in** range(2, int(math.sqrt(limit)) **+** 1):
8. **if** sieve[p]:
9. primes.append(p)
10. **for** i **in** range(p **\*** p, limit **+** 1, p):
11. sieve[i] **=** False
12. **for** p **in** range(max(2, int(math.sqrt(limit)) **+** 1), limit **+** 1):
13. **if** sieve[p]:
14. primes.append(p)
15. **return** primes
17. @staticmethod
18. **def** palindrome\_machine(start, end):
19. palindromes **=** []
20. **for** length **in** range(len(str(start)), len(str(end)) **+** 1):
21. **if** length **==** 1:
22. **for** num **in** range(max(2, start), min(10, end) **+** 1):
23. **if** num **in** {2, 3, 5, 7}:
24. palindromes.append(num)
25. **elif** length **%** 2 **==** 0:
26. **for** num **in** range(10 **\*\*** (length **//** 2 **-** 1), 10 **\*\*** (length **//** 2)):
27. palindrome **=** int(str(num) **+** str(num)[::**-**1])
28. **if** start <**=** palindrome <**=** end **and** palindrome > 10:
29. palindromes.append(palindrome)
30. **else**:
31. **for** num **in** range(10 **\*\*** (length **//** 2 **-** 1), 10 **\*\*** (length **//** 2)):
32. **for** middle **in** range(10):
33. palindrome **=** int(str(num) **+** str(middle) **+** str(num)[::**-**1])
34. **if** start <**=** palindrome <**=** end **and** palindrome > 10:
35. palindromes.append(palindrome)
36. **return** palindromes

### **Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1, 2\_000 | 20: [2, 3, 5] ... [797, 919, 929] | 0.000 |
| 2 | 100, 10\_000 | 15: [101, 131, 151] ... [797, 919, 929] | 0.001 |
| 3 | 20\_000, 80\_000 | 48: [30103, 30203, 30403] ... [79397, 79697, 79997] | 0.001 |
| 4 | 100\_000, 2\_000\_000 | 190: [1003001, 1008001, 1022201] ... [1993991, 1995991, 1998991] | 0.006 |
| 5 | 2\_000\_000, 9\_000\_000 | 327: [3001003, 3002003, 3007003] ... [7985897, 7987897, 7996997] | 0.010 |
| 6 | 10\_000\_000, 100\_000\_000 | 0: [] | 0.033 |
| 7 | 100\_000\_000, 400\_000\_000 | 2704: [100030001, 100050001, 100060001] ... [399737993, 399767993, 399878993] | 0.300 |
| 8 | 1\_100\_000\_000, 15\_000\_000\_000 | 5474: [10000500001, 10000900001, 10001610001] ... [14998289941, 14998589941, 14998689941] | 4.155 |
| 9 | 15\_000\_000\_000, 100\_000\_000\_000 | 36568: [15001010051, 15002120051, 15002320051] ... [99998189999, 99998989999, 99999199999] | 56.990 |
| 10 | 1, 1\_000\_000\_000\_000 | 47995: [2, 3, 5] ... [99998189999, 99998989999, 99999199999] | 62.403 |

### **Student 2: Nellikkaparambu Mohammed, Mohammed Shuhaib**

**Understanding of the Problem and approach to solving it**

The main focus was to address three key objectives. One of these objectives involved **identifying special numbers** that are both **palindromic and prime**. The program's task was to **specifically count within a given user input range of positive values**, **m** and **n**, and then display the total number of special numbers found.

Furthermore, if the count of special numbers were **below six within the specified range,** the program would display all of them. However, if there were five or more special numbers found, it would display the first and last three in the list.

My main approach involved creating a function called "**is\_prime**” to determine whether a **given number is prime**, employing conventional methods like **trial division** or **more advanced methods like sieving with numpy implementation** (J. Murphy, 2023).

Additionally, I aimed to develop another function solely for **checking if a number is palindromic**, using a simple approach such as comparing the number's string representation with its reverse (**str(n) == str(n)[::-1]**).

Since hardcoded values are not permitted, my primary plan for getting inputs for **m** and **n** involves obtaining **validating user inputs** to ensure they were positive integers. I planned to use **conditional statements**, such as IF statements, to handle exceptions, like printing entire list if the length of the list was below six.

For cases where more than five special numbers were found within the given range of m and n, the plan was to use **common python indexing** to access both **the first and last three elements in the list [::3] & [::-3]** .

**Description of code 1 (Original Solution)**

The main difference is visible in the function’s logic, the **is\_prime() function** checks whether a given integer [**n is a prime number**](https://www.geeksforgeeks.org/prime-numbers/). It uses a **basic primality test algorithm that first checks for standard base cases**: if input n is **less than or equal to 1, it returns False; if n is less than or equal to 3 [2 or 3], it returns True** due to prime number properties in general.

In this original **is\_prime** function, the **trial division method** was chosen as an appropriate solution due to it looping up to the **square root of n**, checking if n is divisible by any numbers in the form of **6k ± 1** (Quora, 2019 (5))**,** excluding multiples of 2 and 3 for faster computation. If n is divisible by any such number, it returns False, indicating that n is not prime number. Otherwise, it returns True if n is a prime number. This is\_prime primality check, is quite effective especially for relatively small integers but this code does have its limitation as we move onto bigger test cases.

**Example of Prime elimination (Similar to the popular Eratosthenes Algorithm)**

A table of numbers with a white background

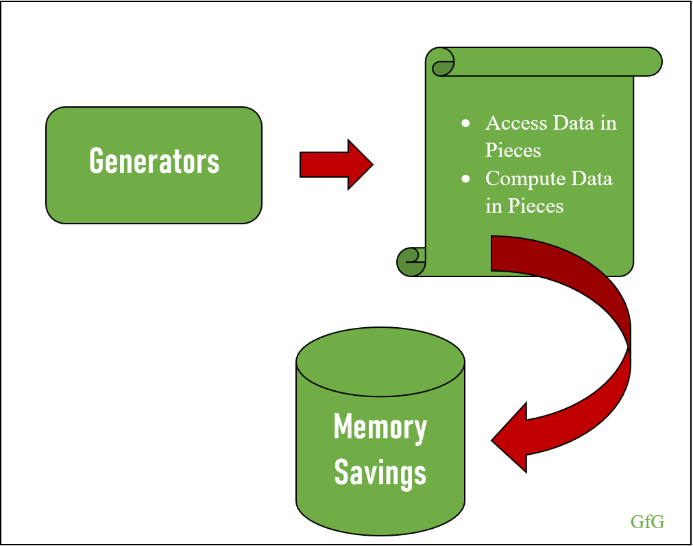
Description automatically generated

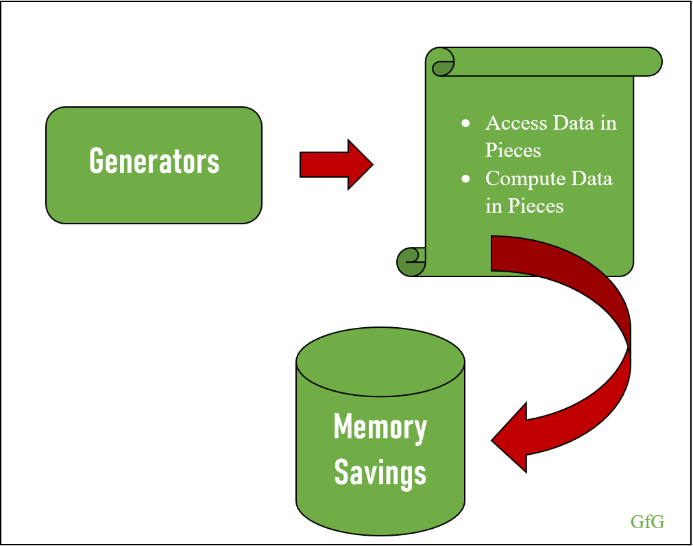
*Trial Division Elimination Exemplar*. (2007, December 27).

As seen above, the image indicates how prime numbers are circled and subsequently their multiples are marked / eliminated from the primality check to fasten up computation.

This primality checking algorithm iterates from **constant 5** up to the square root of the given input 'n'. This approach primarily was chosen for code efficiency reasons. While the standard approach typically checks up to **'sqrt(n) - 1'** (Kumar, 2023). During research, it was found that it doesn't significantly affect the accuracy of the output and it is quicker to simply check all possible divisors within this range, hence we only iterate up to sqrt(n). If none are found, then the number is deemed prime or ‘**True’**.

Another difference is in the palindrome function’s logic, the pali\_maker() function manages most of the palindrome calculations in the program, producing them when required. Instead of the traditional 'return' statement, the **'yield'** keyword was used for a specific purpose. Given that complex generators in Python might face memory limitation / issues when handling bigger inputs, **'yield'** is used to prevent storing all data simultaneously in memory.





*Python Generator Yield Exemplar*. (2020). Retrieved February 20, 2023,.

This method computes and increments on the fly, rather than **pre-calculating everything**, thereby improving palindrome processing efficiency and minimising unnecessary memory usage by storing values beforehand.

As for the logic within the pali\_maker function, it initially yields 0 and then proceeds to yield all single-digit palindromes by iterating within a for loop. Subsequently, it generates the first half of the palindromes using the logic (range(2,20)), which determines the length of the palindrome and calculates it accordingly.

The program is designed to convert / cast the calculated first half into a string for easier manipulation for subsequent steps in determining the other half of the palindrome. A straightforward conditional statement checks if the length of the constant obtained earlier is even by calculating its modulo with 2.

If the remainder is 0, the first half of the palindrome is simply mirrored [3] and yielded as a palindrome. Otherwise, the program enters the else statement due to the length being odd. In this case, it iterates over the range 0 to 9 to cover all possible middle digits, reverses the first half, combines it with the middle digit, and appends the mirrored first half, yielding the resulting palindrome back as an integer for primality checking afterwards.

The appender function collates all the data provided above. It initialises an empty list called "special\_found" to store calculated values. It invokes the pali\_maker function to generate palindromes. A conditional statement is applied to check whether the generated number, determined by the pali\_maker function, is prime using the is\_prime() function, and falls within the specified range defined by "m" and "n".

If this condition is met, the palindrome is appended to the empty list. The pali\_maker function is iteratively called to obtain the next palindrome to check these conditionals once again until it has reached the end. Finally, the appender function returns the "special\_found" list.

**Difference in Code [Highlighted Section]**

1. **def** is\_prime(n):
2. **if** n <**=** 1:
3. **return** False
4. **if** n <**=** 3:
5. **return** True
6. **if** n **%** 2 **==** 0 **or** n **%** 3 **==** 0:
7. **return** False
9. max\_divisor **=** math.isqrt(n) **+** 1
10. **for** i **in** range(5, max\_divisor, 6):  # here I'm applying a form of trial division as primes over 3 in form 6k +- 1
11. **if** n **%** i **==** 0 **or** n **%** (i **+** 2) **==** 0:
12. **return** False
14. **return** True

17. **def** pali\_maker():
18. **yield** 0
19. **for** i **in** range(1, 10):
20. **yield** i
21. **for** constant **in** range(2, 20):
22. **for** half **in** range(10 **\*\*** (constant **//** 2 **-** 1), 10 **\*\*** (constant **//** 2)):
23. half\_str **=** str(half)
24. **if** constant **%** 2 **==** 0:
25. **yield** int(half\_str **+** half\_str[::**-**1])
26. **else**:
27. **for** mid **in** range(10):
28. **yield** int(half\_str **+** str(mid) **+** half\_str[::**-**1])

### **Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input [Format : (m, n)]** | **Output** | **Running time (s)** |
| 1 | (1, 2\_000) | 20: [2, 3, 5, 797, 919, 929] | 0.000 |
| 2 | (100, 10000) | 15: [101, 131, 151, 797, 919, 929] | 0.000 |
| 3 | (20000, 80000) | 48: [30103, 30203, 30403, 79397, 79697, 79997] | 0.005 |
| 4 | (100000, 2000000) | 190: [1003001, 1008001, 1022201, 1993991, 1995991, 1998991] | 0.016 |
| 5 | (2\_000\_000, 9\_000\_000) | 327: [3001003, 3002003, 3007003, 7985897, 7987897, 7996997] | 0.015 |
| 6 | (10\_000\_000, 100\_000\_000) | 0: [] | 0.044 |
| 7 | (100\_000\_000, 400\_000\_000) | 2704: [100030001, 100050001, 100060001, 399737993, 399767993, 399878993] | 1.187 |
| 8 | (1\_100\_000\_000, 15\_000\_000\_000) | 5474: [10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941] | 0.967 |
| 9 | (15\_000\_000\_000, 100\_000\_000\_000) | 36568: [15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999] | 164.035 |
| 10 | (1, 1\_000\_000\_000\_000) | 47995: [2, 3, 5, 99998189999, 99998989999, 99999199999] | 254.587 |
| \*Running time results may vary depending on machine specifications. | | | |

### **Student 3: Lemoudden, Ilyass**

**Understanding of the Problem and approach to solving it**

In this code, the **generate\_palindromic\_numbers** function creates palindromic numbers within a range. It contains a nested function called **generate\_palindromes** that specifically makes these numbers. It makes these numbers by finding out how many digits are in the number its checking to see if it’s an odd number, or an even number. If the length is even, it will mirror the left half of the number to form the right half. However, if the length is odd, it will add a number from 0 to 9 in the middle before mirroring the left half of the number. In this function, the code also starts and ends the timer so that the total time taken to find all the special numbers in the given range can be measured. It does this by using the built-in function time, and importing it in.

The **is\_prime** function is designed to determine whether the given number, which are already palindromes are prime numbers. It begins by checking if the number is less than 2, as numbers less than 2 are not considered prime. If the number is equal to 2, it is identified as prime since 2 is the only even prime number. The function then proceeds to check if the number is even, as all even numbers greater than 2 are not prime. the function then iterates through odd numbers starting from 3 up to the square root of the given number checking for divisibility, if any are found, then the number is deemed not prime, and the code returns false, otherwise, it will return true.

The **special\_numbers** function is designed to identify and print prime palindromic numbers within a specified range. It begins by initializing an empty list to store the prime palindromic numbers found within the given range. The function then uses the **generate\_palindromic\_numbers** function to generate palindromic numbers within the specified range, it then uses the **is\_prime** function to filter out the nonprime palindromes by checking each palindromic number generated with the **is\_prime** function. If a palindromic number is also a prime number, it is appended to the list of special numbers. Once every palindromic number is tested to see if its prime, and every palindromic prime number has been appended into the special numbers list, the code then checks to see how many special numbers are in the list. If there are less than six palindromic prime, the function simply prints the list of special numbers. However, if there are six or more palindromic prime numbers, the function prints the total length of the list to tell us how many special numbers were found, followed by the first three and last three numbers in the list.

**Difference in Code [Highlighted Section]**

1. **def** generate\_palindromic\_numbers(start, end):
2. **def** generate\_palindromes(digits):
4. **if** digits **==** 1:
5. **for** i **in** range(1, 10):
6. **yield** i
7. **elif** digits **%** 2 **==** 0:
8. half **=** digits **//** 2
9. half\_range **=** range(10 **\*\*** (half **-** 1), 10 **\*\*** half)
10. **for** i **in** half\_range:
11. palindrome **=** int(str(i) **+** str(i)[::**-**1])
12. **if** start <**=** palindrome <**=** end:
13. **yield** palindrome
14. **else**:
15. half **=** digits **//** 2
16. half\_range **=** range(10 **\*\*** (half **-** 1), 10 **\*\*** half)
17. **for** i **in** half\_range:
18. **for** j **in** range(10):
19. palindrome **=** int(str(i) **+** str(j) **+** str(i)[::**-**1])
21. **if** start <**=** palindrome <**=** end:
22. **yield** palindrome
24. start\_digits **=** len(str(start))
25. end\_digits **=** len(str(end))
26. start\_time **=** time.time()
27. **for** digits **in** range(start\_digits, end\_digits **+** 1):
28. **yield** **from** generate\_palindromes(digits)
30. end\_time **=** time.time()
31. print("Time taken:", end\_time **-** start\_time, "seconds")

### **Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1, 2\_000 | Length: 20  2, 3, 5, 797, 919, 929 | 0.000 |
| 2 | 100, 10\_000 | Length: 15  101, 131, 151, 797, 919, 929 | 0.009 |
| 3 | 20\_000, 80\_000 | Length: 48  30103, 30203, 30403, 79397, 79697, 79997 | 0.009 |
| 4 | 100\_000, 2\_000\_000 | Length: 190  1003001, 1008001, 1022201, 1993991, 1995991, 1998991 | 0.010 |
| 5 | 2\_000\_000, 9\_000\_000 | Length: 327  3001003, 3002003, 3007003, 7985897, 7987897, 7996997 | 0.020 |
| 6 | 10\_000\_000, 100\_000\_000 | Length: 0 | 0.059 |
| 7 | 100\_000\_000, 400\_000\_000 | Length: 2704  100030001, 100050001, 100060001, 399737993, 399767993, 399878993 | 1.610 |
| 8 | 1\_100\_000\_000, 15\_000\_000\_000 | Length: 5474  10000500001, 10000900001, 10001610001, 14998289941, 14998589941, 14998689941 | 24.200 |
| 9 | 15\_000\_000\_000, 100\_000\_000\_000 | Length: 36568  15001010051, 15002120051, 15002320051, 99998189999, 99998989999, 99999199999 | 335.70 |
| 10 | 1, 1\_000\_000\_000\_000 | Length: 47995  2, 3, 5, 99998189999, 99998989999, 99999199999 | 375.100 |

### **Student 4: Ali, Wais**

**Understanding of the Problem and approach to solving it**

The proposed python code is an interesting tool for finding numbers that are both prime and palindromic. To do this, it consists of multiple finely crafted components. The function that determines whether a given number is prime is the first one primic(); it uses effective algorithm to handle a variety of scenarios and maximise efficiency. In a similar vein, another function checks whether a given number reads the same both forward and backward is the palindromic(). The script also includes a way to find these unique numbers within the given range, which simplifies the process and eliminates needless calculations. The code shows strong performance and fast results when handling ranges from 1 to hundreds of millions. However, as it gets closer to very large ranges, like a billion or a trillion, its efficiency starts to decrease. The fundamental complexity of prime number testing at such large scales is the cause of this slowdown. Even with the optimised algorithms included in the code, there are just so many numbers to test in these large ranges, which naturally takes more time and computational power. Nonetheless, the script is still a useful computational and educational tool, especially useful for showing the ideas of prime numbers and palindromes in large but manageable numerical ranges.

**Difference in Code [Highlighted Section]**

1. **def** primic(num):
2. **if** num < 2:
3. **return** False
4. **if** num **==** 2:
5. **return** True
6. **if** num **%** 2 **==** 0:
7. **return** False
8. **for** i **in** range(3, int(num **\*\*** 0.5) **+** 1, 2):
9. **if** num **%** i **==** 0:
10. **return** False
11. **return** True

14. **def** palindromic(num):
15. **return** str(num) **==** str(num)[::**-**1]

18. **def** fetch\_special\_numbers(m, n):
19. special\_numbers **=** []
20. **for** number **in** range(max(2, m), n **+** 1):
21. **if** primic(number) **and** palindromic(number):
22. special\_numbers.append(number)
23. **return** special\_numbers

### **Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1, 2\_000 | Special numbers between 1 and 2000: [2, 3, 5, 797, 919, 929]  Total special numbers found: 20 | 0.00s |
| 2 | 100, 10\_000 | Special numbers between 100 and 10000: [101, 131, 151, 797, 919, 929]  Total special numbers found: 15 | 0.00s |
| 3 | 20\_000, 80\_000 | Special numbers between 20000 and 80000: [30103, 30203, 30403, 79397, 79697, 79997]  Total special numbers found: 48 | 0.04s |
| 4 | 100\_000, 2\_000\_000 | Special numbers between 100000 and 2000000: [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]  Total special numbers found: 190 | 3.93s |
| 5 | 2\_000\_000, 9\_000\_000 | Special numbers between 2000000 and 9000000: [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]  Total special numbers found: 327 | 29.13s |
| 6 | 10\_000\_000, 100\_000\_000 | Special numbers between 10000000 and 100000000: []  Total special numbers found: 0 | 1151.29s |
| 7 | 100\_000\_000, 400\_000\_000 | NA | >3600s |
| 8 | 1\_100\_000\_000, 15\_000\_000\_000 | NA | >3600s |
| 9 | 15\_000\_000\_000, 100\_000\_000\_000 | NA | >3600s |
| 10 | 1, 1\_000\_000\_000\_000 | NA | >3600s |

### **Student 5: Hernandez, Zak**

**Understanding of the Problem and approach to solving it**

The approach here aims to identify and count special numbers within specified ranges. A special number is defined as a prime number which is also a palindrome. The code comprises three main functions: **primic, palindromic, and look\_for\_special\_numbers.**

The **primic** function checks whether a given number is prime, utilising basic primality test conditions. It first verifies if the number is less than 2, returning False if so. For numbers 2 and above, it iterates through potential divisors up to the square root of the number, checking for divisibility. If the number then passes these conditions, it returns True, indicating the primality.

The **palindromic** function checks if a number is a palindrome by converting them to a string and comparing it with its reversed string. If the number reads the same forwards and backwards, it returns True; otherwise, it returns False.

The **look\_for\_special\_numbers** function generates a list of special numbers within a given range. It iterates through each number in the range, checking if it satisfies both primality and palindrome conditions. The special numbers are stored in a list, and depending on the count, either all special numbers or the first and last three are printed along with the total count.

The test cases provided cover various ranges, allowing for the evaluation of the code's performance across different input scenarios. Each test case is executed, and the special numbers within the specified range are printed along with the total length of number of special numbers.

Overall, the code efficiently identifies special numbers within specified ranges by employing primality and palindrome checks, providing a clear cut solution for the problem statement in the specifications.

**Difference in Code [Highlighted Section]**

1. **def** primic(number):
2. **if** number < 2:
3. **return** False
4. **elif** number **==** 2:
5. **return** True
6. **elif** number **%** 2 **==** 0:
7. **return** False
8. **else**:
10. **for** i **in** range(3, int(number**\*\***0.5)**+**1, 2):
11. **if** number **%** i **==** 0:
12. **return** False
13. **return** True

16. **def** palindromic(number):
17. **if** str(number) **==** str(number)[::**-**1]:
18. **return** True
19. **else**:
20. **return** False

### **Results**

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Running time (s)** |
| 1 | 1, 2\_000 | [2, 3, 5, 797, 919, 929]  Amount of Special Numbers: 20 | 0.000 |
| 2 | 100, 10\_000 | [101, 131, 151, 797, 919, 929]  Amount of Special Numbers: 15 | 0.007 |
| 3 | 20\_000, 80\_000 | [30103, 30203, 30403, 79397, 79697, 79997]  Amount of Special Numbers: 48 | 0.059 |
| 4 | 100\_000, 2\_000\_000 | [1003001, 1008001, 1022201, 1993991, 1995991, 1998991]  Amount of Special Numbers: 190 | 5.921 |
| 5 | 2\_000\_000, 9\_000\_000 | [3001003, 3002003, 3007003, 7985897, 7987897, 7996997]  Amount of Special Numbers: 327 | 49.913 |
| 6 | 10\_000\_000, 100\_000\_000 | []  Amount of Special Numbers: 0 | 1863.814 |
| 7 | 100\_000\_000, 400\_000\_000 | NA | >3600 |
| 8 | 1\_100\_000\_000, 15\_000\_000\_000 | NA | >3600 |
| 9 | 15\_000\_000\_000, 100\_000\_000\_000 | NA | >3600 |
| 10 | 1, 1\_000\_000\_000\_000 | NA | >3600 |

# Test and analyse your solution!

**Test Cases Selection and Justification**

Judging from the vast diversity of efficient codes and carefully crafted techniques of each of the group members’ codes, this part of the report was decided to be agreed upon custom crafted 10 valid and positive range taste cases ranging from the lower bound value of 1 and incrementing exponentially, first three being small range cases for testing low range accuracy, next 2 being mid-range cases to test palindrome mirroring efficiency for odd and even digits next 3 being high end exponential cases incrementally increasing with the expression 10^(n+1) where n = 10 and progresses as n = n + 1. The final 3 are meticulously crafted trivial test cases bound to be tested with negative, invalid, and non-sensical values for code accuracy and efficiency.

### 

### **Test cases:**

1st Test Case – 1 to 10^2

2nd Test Case – 1 to 500

3rd Test Case – 1 to 10^3

4th Test Case – 1 to 10^4

5th Test Case – 1 to 10^6

6th Test Case – 1 to 10^11

7th Test Case – 1 to 10^12

8th Test Case – 1 to 10^13

9th Test Case – (-10^3) to 10^3

10th Test Case – (5.5) to (2000.5)

11th Test Case – 10^7 to 10^8

### **Student 1’s (Raut, Nihaar) Analysis:**

**Test Case Table and Justification with Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **c** | **Input** | **Output** | **Justification** | **Student 1 results** |
| 1 | 1 to 100 | 5: [2, 3, 5, 7, 11]  Test case 1 completed in 0.000 seconds | Small range to calculate special numbers < 6 | 0.000s |
| 2 | 1 to 500 | 14: [2, 3, 5] ... [353, 373, 383]  Test case 2 completed in 0.000 seconds | Another small range to calculate special numbers up to two digits. | 0.000s |
| 3 | 1 to 1\_000 | 20: [2, 3, 5] ... [797, 919, 929]  Test case 3 completed in 0.000 seconds. | Testing limits of small range with 10^3 to get special numbers parallel to one of the given test cases. | 0.000s |
| 4 | 1 to 10\_000 | 20: [2, 3, 5] ... [797, 919, 929]  Test case 4 completed in 0.000 seconds. | Mid-range testing for 4th test case but same result as previous signalling no extra special numbers. | 0.000s |
| 5 | 1 to 1\_000\_000 | 113: [2, 3, 5] ... [97879, 98389, 98689]  Test case 5 completed in 0.005 seconds. | Increasing mid-range test case size to stretch special numbers to three digits. | 0.005s |
| 6 | 1 to 100\_000\_000\_000 | 47995: [2, 3, 5] ... [99998189999, 99998989999, 99999199999]  Test case 6 completed in 66.241 seconds. | Evolving to high end test case results for large scale answers and efficiency checking. | 66.241s |
| 7 | 1 to 1\_000\_000\_000\_000 | 47995: [2, 3, 5] ... [99998189999, 99998989999, 99999199999]  Test case 6 completed in 82.010 seconds. | Stopping at 10^12 since no new special numbers between 10^11 and 10^12 for accuracy. | 82.010s |
| 8 | 1 to 10\_000\_000\_000\_000 | 401696: [2, 3, 5] ... [9999970799999, 9999980899999, 9999987899999]  Test case 10 completed in 901.121 seconds. | Testing high-end limit for 10^13 test case which has 6-digit special numbers. | 901.121s |
| - | - | - | - | - |
| 9 | (-1\_000) to 1\_000 | 0: [] Test case 10 completed in 0.000 seconds. | Testing a curated negative to positive range, results expected, doesn’t function for negative to positive ranges. | 0.000s |
| 10 | 5.5 to 2000.5 | 15: [101, 131, 151] ... [797, 919, 929] Test case 10 completed in 0.000 seconds. | Testing between float ranges and code does work between the decimals. | 0.000s |
| 11 | 10\_000\_000 to 100\_000\_000 | 0: [] Test case 10 completed in 0.033 seconds. | Testing an empty range which does yield accurate result. | 0.033s |

### 

### **Running time graphs**

**Graphical Representation of the Results**

A graph with a line and a blue line

Description automatically generated

*The graph shows the performance of the code with the selected test cases in the above table*

### **Complexity analysis**

Looking at the runtime vs Test Case input graph, it’s possible to move forward and calculate the Big-O notation of the proposed code.

**Prime Sieve Algorithm:** The **'prime\_siever'** method implements the segmented Sieve of Eratosthenes algorithm to generate prime numbers up to the square root of the given limit. The time complexity of this algorithm is **O(nlog(log(n)),** where n is the limit.

**Palindrome Generation:** The **'palindrome\_machine'** method generates palindromic numbers within the given range. It loops over the length of the numbers in the range and generates odd length and even length palindromes accordingly. The time complexity of this part depends on the length of the range and the length of the numbers in the range. Since it generates palindromes up to the square of the end value, its time complexity can be considered as **O(√N),** where N is the end value.

Considering these components, the overall time complexity of the code can be found out as

the maximum among the time complexities of its parts. Therefore, final complexity of the code is approximately **O(√N log(log(√N))),** where N is the end value of the range.

### **Student 2’s (Nellikkaparambu Md. Md. Shuhaib) Analysis:**

### **Student 3’s (Lemoudden, Ilyass) Analysis:**

**Test Case Table and Justification with Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **c** | **Input** | **Output** | **Justification** | **Student 3 results** |
| 1 | 1 to 100 | Time taken: 0.0 seconds Length: 5  [2, 3, 5, 7, 11] | Small range to calculate special numbers < 6 | 0.000s |
| 2 | 1 to 500 | Time taken: 0.0 seconds  Length: 14  2, 3, 5, 353, 373, 383 | Another small range to calculate special numbers up to two digits. | 0.000s |
| 3 | 1 to 1\_000 | Time taken: 0.0 seconds  Length: 20  2, 3, 5, 797, 919, 929 | Testing limits of small range with 10^3 to get special numbers parallel to one of the given test cases. | 0.000s |
| 4 | 1 to 10\_000 | Time taken: 0.001 seconds  Length: 20  2, 3, 5, 797, 919, 929 | Mid-range testing for 4th test case but same result as previous signalling no extra special numbers. | 0.001s |
| 5 | 1 to 1\_000\_000 | Time taken: 0.005 seconds  Length: 113  2, 3, 5, 97879, 98389, 98689 | Increasing mid-range test case size to stretch special numbers to three digits. | 0.005s |
| 6 | 1 to 100\_000\_000\_000 | Time Taken: 356.222 seconds Length: 47995  2, 3, 5, 99998189999, 99998989999, 99999199999 | Evolving to high end test case results for large scale answers and efficiency checking. | 356.222s |
| 7 | 1 to 1\_000\_000\_000\_000 | Time Taken: 391.242 seconds Length: 47995  2, 3, 5, 99998189999, 99998989999, 99999199999 | Stopping at 10^12 since no new special numbers between 10^11 and 10^12 for accuracy. | 391.242s |
| 8 | 1 to 10\_000\_000\_000\_000 | NA | Testing high-end limit for 10^13 test case which has 6-digit special numbers. | >3600s |
| - | - | - | - | - |
| 9 | (-1\_000) to 1\_000 | Time taken: 0.0 seconds  [] | Testing a curated negative to positive range, results expected, doesn’t function for negative to positive ranges. | 0.000s |
| 10 | 5.5 to 2000.5 | Error | Testing between float ranges and code doesn’t work between the decimals. | NA |
| 11 | 10\_000\_000 to 100\_000\_000 | Time Taken: 0.059 seconds Length: 0 | Testing an empty range which does yield accurate result. | 0.059s |

### **Running time graphs**

**Graphical Representation of the Results**

A graph with a line going up

Description automatically generated

*The graph shows the performance of the code with the selected test cases in the above table*

### **Complexity analysis**

The code generates palindromic numbers within a specified given range and identifies special numbers in a defined complexity:

**Generating Palindromic Numbers:**

The function generated palindromes based on the number of digits in the specified range,

generating them using nested loops and conditions. The time complexity is around O(n √m), where n is the number of digits in the range and m is the highest value in the range.

**Checking Prime Numbers:**

This method checks whether a given number is prime by looping through odd numbers up to the square root of the number. The time complexity of checking primality Is around O(√n), where n is the given number.

**Identifying Special Numbers:**

After generating palindromic numbers, the code verifies each one checking for primality to identify special numbers.

The overall time complexity of identifying special numbers is approximately **O(n √n)**,

where n is the highest value in the specified range.

### **Student 4’s (Ali, Wais) Analysis:**

**Test Case Table and Justification with Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **c** | **Input** | **Output** | **Justification** | **Student 4 results** |
| 1 | 1 to 100 | Special numbers between 1 and 100: [2, 3, 5, 7, 11]  Total special numbers found: 5  Time taken: 0.00 seconds | Small range to calculate special numbers < 6 | 0.000s |
| 2 | 1 to 500 | Special numbers between 1 and 500: [2, 3, 5, 353, 373, 383]  Total special numbers found: 14  Time taken: 0.00 seconds | Another small range to calculate special numbers up to two digits. | 0.000s |
| 3 | 1 to 1\_000 | Special numbers between 1 and 1000: [2, 3, 5, 797, 919, 929]  Total special numbers found: 20  Time taken: 0.00 seconds | Testing limits of small range with 10^3 to get special numbers parallel to one of the given test cases. | 0.000s |
| 4 | 1 to 10\_000 | Special numbers between 1 and 10000: [2, 3, 5, 797, 919, 929]  Total special numbers found: 20  Time taken: 0.01 seconds | Mid-range testing for 4th test case but same result as previous signalling no extra special numbers. | 0.001s |
| 5 | 1 to 1\_000\_000 | Special numbers between 1 and 1000000: [2, 3, 5, 97879, 98389, 98689]  Total special numbers found: 113  Time taken: 1.02 seconds | Increasing mid-range test case size to stretch special numbers to three digits. | 1.020s |
| 6 | 1 to 100\_000\_000\_000 | NA | Evolving to high end test case results for large scale answers and efficiency checking. | >3600s |
| 7 | 1 to 1\_000\_000\_000\_000 | NA | Stopping at 10^12 since no new special numbers between 10^11 and 10^12 for accuracy. | >3600s |
| 8 | 1 to 10\_000\_000\_000\_000 | NA | Testing high-end limit for 10^13 test case which has 6-digit special numbers. | >3600s |
| - | - | - | - | - |
| 9 | (-1\_000) to 1\_000 | Special numbers between -1000 and 1000: [2, 3, 5, 797, 919, 929]  Total special numbers found: 20  Time taken: 0.00 seconds | Testing a curated negative to positive range, results expected, does function for negative to positive ranges. | 0.000s |
| 10 | 5.5 to 2000.5 | Error | Testing between float ranges and code doesn’t work between the decimals. | NA |
| 11 | 10\_000\_000 to 100\_000\_000 | Special numbers between 10000000 and 100000000:  Total special numbers found: 0  Time taken: 1151.29 seconds | Testing an empty range which does yield accurate result. | 1151.29s |

### **Running time graphs**

**Graphical Representation of the Results**

A graph with a line

Description automatically generated

*The graph shows the performance of the code with the selected test cases in the above table*

### **Complexity analysis**

The code shows two main functions:

**'primic'** and **'palindromic'**, responsible for determining whether a number is prime and whether it's a palindrome, respectively. The 'primic' function efficiently checks for primality using trial division up to the square root of the number, omitting even numbers after considering 2 as a special case. Whereas the **'palindromic**' function

Takes in consideration string manipulation to compare a number with its reversed string representation, efficiently determining its palindromic property. These functions are then utilised within the **`fetch\_special\_numbers'** function, which loops through the range specified by the input of the user, identifying and appending special numbers to a

list. Ultimately, the **'display\_numbers'** function ensures that only a set of special numbers, either all or just the first and last three, is displayed based on the count of special numbers found. The time complexity of the code is **O(n √n)**, where n represents the size of the range between the smaller and larger numbers.

### **Student 5’s (Hernandez, Zak) Analysis:**

**Test Case Table and Justification with Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **c** | **Input** | **Output** | **Justification** | **Student 4 results** |
| 1 | 1 to 100 | Test case 1:  [2, 3, 5, 7, 11]  Amount of Special Numbers: 5 | Small range to calculate special numbers < 6 | 0.000s |
| 2 | 1 to 500 | Test case 2:  [2, 3, 5, 353, 373, 383]  Amount of Special Numbers: 14 | Another small range to calculate special numbers up to two digits. | 0.000s |
| 3 | 1 to 1\_000 | Test case 3:  [2, 3, 5, 797, 919, 929]  Amount of Special Numbers: 20 | Testing limits of small range with 10^3 to get special numbers parallel to one of the given test cases. | 0.001s |
| 4 | 1 to 10\_000 | Test case 4:  [2, 3, 5, 797, 919, 929]  Amount of Special Numbers: 20 | Mid-range testing for 4th test case but same result as previous signalling no extra special numbers. | 0.002s |
| 5 | 1 to 1\_000\_000 | Test case 5:  [2, 3, 5, 97879, 98389, 98689]  Amount of Special Numbers: 113 | Increasing mid-range test case size to stretch special numbers to three digits. | 0.059s |
| 6 | 1 to 100\_000\_000\_000 | NA | Evolving to high end test case results for large scale answers and efficiency checking. | >3600s |
| 7 | 1 to 1\_000\_000\_000\_000 | NA | Stopping at 10^12 since no new special numbers between 10^11 and 10^12 for accuracy. | >3600s |
| 8 | 1 to 10\_000\_000\_000\_000 | NA | Testing high-end limit for 10^13 test case which has 6-digit special numbers. | >3600s |
| - | - | - | - | - |
| 9 | (-1\_000) to 1\_000 | Test case 1:  [2, 3, 5, 797, 919, 929]  Amount of Special Numbers: 20 | Testing a curated negative to positive range, results expected, does  function for negative to positive ranges. | 0.000s |
| 10 | 5.5 to 2000.5 | Error | Testing between float ranges and code doesn’t work between the decimals. | NA |
| 11 | 10\_000\_000 to 100\_000\_000 | Test case 1:  []  Amount of Special Numbers: 0 | Testing an empty range which does yield accurate result. | 1868.299s |

### **Running time graphs**

**Graphical Representation of the Results**

A graph with a line and numbers

Description automatically generated

*The graph shows the performance of the code with the selected test cases in the above table*

### **Complexity analysis**

The time complexity of proposed code, O(n √n), primarily stems from the looping through the rangeof numbers, which scales linearly with the size of the range (n).

Within each iteration, the **‘primic’** function is called to check if the current number is prime. This function then loops up to the square root of the current number, resulting in a time complexity of O(√n) for each number in the range.

Similarly, the **‘palindromic’** function is brought in to determine if the number is a palindrome, which involves conversion of the number to a string and comparing it with

its reverse, leading to a constant time complexity operation. T

herefore, for each number in the range, both the primic and palindromic functions contribute O(√n) operations, resulting in a combined time complexity of **O(n√n)** for the overall code.

### **Overall Running time graphs**

**Graphical Representation of the Overall Group Results**

# Optimise solutions!

Include the final code with outputs and running time measurements in the report’s Appendix section and also upload it for Deliverable 2.

Remember, your optimisation goal is to solve correctly as many test cases as possible, as shown in the test case table above.)

**Choice of codes to optimise further and explanation**

Student 1’s (Raut, Nihaar) and Student 2’s (Mohammed, Shuhaib’s) Code were chosen collectively to optimise further and improve the time & space complexities along with the overall accuracy. These codes were chosen because of the potential of improvement of prime checking for Raut, Nihaar’s code and Mohammed, Shuhaib’s code. Both individuals believe the prime checking and generation could be further improved and the palindrome checking for primality and furthermore greatly reduce in time. The palindrome generation methods however, stay the same in both codes after [diagnosing slow code](https://www.youtube.com/watch?v=m_a0fN48Alw&pp=ygUSZGlhZ25vc2Ugc2xvdyBjb2Rl) gave away that primality checking was the only way to decrease the time complexity and further improve the overall algorithm efficiently for executing the given test cases.

### **Solution 1 (Raut, Nihaar):**

**IMPROVEMENT REPORT**

**What did the student do to improve on their previous code?**

The replacement here was with the segmented sieve of Eratosthenes algorithm, used for prime number generation, with the Miller-Rabin probabilistic primality test since this was found way more compatible and efficient for my code.

**Why did they improve it?**

The previous code utilised the segmented sieve of Eratosthenes, which is effective for generating a large list of prime numbers but has limitations when it comes to large ranges, leading to relatively slower execution. The Miller-Rabin algorithm is probabilistic but significantly more accurate and efficient when paired with k values > 2, it is faster for primality testing, especially for large numbers or ranges, for instance the ones having upper bound over 10^9. By switching to Miller-Rabin, the aim was shifted the efficiency of the prime number checking process and not touching the palindrome machine since it has already peaked.

**How did they improve it?**

The change involved implementing the Miller-Rabin primality test function prime\_checker instead of the prime\_seiver and is\_prime from the segemented seive. This function performs probabilistic tests on whether a given number is prime. The algorithm involves choosing random witnesses generating throughout the code, achieved after importing random and performing modular exponentiation operations to determine if a number is composite with a high degree of certainty scaling with the initial k value provided. By leveraging this algorithm, the code achieves faster prime number checking compared to the segmented sieve of Eratosthenes and the proof will be avaiable in the underlying test case running table after this section.

**How did they improve it?**

The Miller-Rabin primality test has a fixed time complexity of O(k \* log^3(n)), where k is the number of iterations and n is the number being tested. While k is typically a small constant (set to 3 in the provided code), the time complexity largely depends on the magnitude of the number being tested. In practical scenarios, k=3 provides efficient prime testing for large numbers, including those within the specified ranges in the test cases. The certainty can be further increased or decreased for further larger numbers adjusting the k value desirable to us. It will however come to the cost of runtimes of the codes.

**Overall, by replacing the segmented sieve of Eratosthenes with the Miller-Rabin primality test, this new code achieves faster execution and maintains low time and space complexity, making it very well suitable for the pre-provided test cases and improving the efficiency of prime number checking along with the stagnant palindrom generation with palindrome machine.**

1. **import** time
2. **import** random
3. **def** prime\_checker(num, k**=**3):
4. beginning\_primes **=** [2, 3, 5, 7]
5. **if** num **in** beginning\_primes:
6. **return** True
7. **if** num <**=** 1 **or** any(num **%** p **==** 0 **for** p **in** beginning\_primes):
8. **return** False
9. r, d **=** 0, num **-** 1
10. **while** d **%** 2 **==** 0:
11. r **+=** 1
12. d **//=** 2
13. **for** \_ **in** range(k):
14. a **=** random.randint(2, num **-** 1)
15. x **=** pow(a, d, num)
16. **if** x **==** 1 **or** x **==** num **-** 1:
17. **continue**
18. **for** \_ **in** range(r **-** 1):
19. x **=** pow(x, 2, num)
20. **if** x **==** num **-** 1:
21. **break**
22. **else**:
23. **return** False
24. **return** True

### **Results**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Input** | **Output** | **Correctness** | **Running time (s)** |
| 1 | 1, 2\_000 | 20: [2, 3, 5] ... [797, 919, 929]  Test case 1 completed in 0.001 seconds. | PASS  Equally accurate and efficient | 0.001 |
| 2 | 100, 10\_000 | 15: [101, 131, 151] ... [797, 919, 929]  Test case 2 completed in 0.000 seconds. | PASS  Equally accurate and efficient | 0.000 |
| 3 | 20\_000, 80\_000 | 48: [30103, 30203, 30403] ... [79397, 79697, 79997]  Test case 3 completed in 0.002 seconds. | PASS  Equally accurate and efficient | 0.002 |
| 4 | 100\_000, 2\_000\_000 | 190: [1003001, 1008001, 1022201] ... [1993991, 1995991, 1998991]  Test case 4 completed in 0.008 seconds. | PASS  Equally accurate and efficient | 0.008 |
| 5 | 2\_000\_000, 9\_000\_000 | 327: [3001003, 3002003, 3007003] ... [7985897, 7987897, 7996997]  Test case 5 completed in 0.012 seconds. | PASS  Equally accurate and efficient | 0.012 |
| 6 | 10\_000\_000, 100\_000\_000 | 0: []  Test case 6 completed in 0.040 seconds. | PASS  Equally accurate and efficient | 0.040 |
| 7 | 100\_000\_000, 400\_000\_000 | 2704: [100030001, 100050001, 100060001] ... [399737993, 399767993, 399878993]  Test case 7 completed in 0.102 seconds. | PASS  Equally accurate and marginally more efficient | 0.102 |
| 8 | 1\_100\_000\_000, 15\_000\_000\_000 | 5474: [10000500001, 10000900001, 10001610001] ... [14998289941, 14998589941, 14998689941]  Test case 8 completed in 0.813 seconds. | PASS  Equally accurate and notably efficient | 0.813 |
| 9 | 15\_000\_000\_000, 100\_000\_000\_000 | 36568: [15001010051, 15002120051, 15002320051] ... [99998189999, 99998989999, 99999199999]  Test case 9 completed in 2.937 seconds. | PASS  Equally accurate and highly efficient | 2.937 |
| 10 | 1, 1\_000\_000\_000\_000 | 47995: [2, 3, 5] ... [99998189999, 99998989999, 99999199999]  Test case 10 completed in 8.235 seconds. | PASS  Equally accurate and significantly efficient | 8.235 |

**Runtime graphs**

A graph with numbers and lines

Description automatically generated

### **Solution 2 (Mohammed, Shuhaib):**

# Compare the performance!

### **Detailed Curated Test Case Analysis**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **#** | **Input** | **Student 1’s Output** | **Student 2’s Output** | **Runtime Comparison (s)** |
| 1 | 1, 1000 | 20: [2, 3, 5] ... [797, 919, 929]  Test case 1 completed in 0.001 seconds. |  | S1:0.001  S2: |
| 2 | 1, 10\_000 | 20: [2, 3, 5] ... [797, 919, 929]  Test case 2 completed in 0.000 seconds. |  | S1:0.000  S2: |
| 3 | 1, 100\_000 | 113: [2, 3, 5] ... [97879, 98389, 98689]  Test case 3 completed in 0.003 seconds. |  | S1:0.003  S2: |
| 4 | 1, 1\_000\_000 | 113: [2, 3, 5] ... [97879, 98389, 98689]  Test case 4 completed in 0.006 seconds. |  | S1:0.006  S2: |
| 5 | 1, 10\_000\_000 | 781: [2, 3, 5] ... [9980899, 9981899, 9989899]  Test case 5 completed in 0.023 seconds. |  | S1:0.023  S2: |
| 6 | 1, 100\_000\_000 | 781: [2, 3, 5] ... [9980899, 9981899, 9989899]  Test case 6 completed in 0.061 seconds. |  | S1:0.061  S2: |
| 7 | 1, 1\_000\_000\_000 | 5953: [2, 3, 5] ... [999676999, 999686999, 999727999]  Test case 7 completed in 0.240 seconds. |  | S1:0.240  S2: |
| 8 | 1, 10\_000\_000\_000 | 5953: [2, 3, 5] ... [999676999, 999686999, 999727999]  Test case 8 completed in 0.666 seconds. |  | S1:0.666  S2: |
| 9 | 1, 100\_000\_000\_000 | 47995: [2, 3, 5] ... [99998189999, 99998989999, 99999199999]  Test case 9 completed in 3.515 seconds. |  | S1:3.515  S2: |
| 10 | 1, 1\_000\_000\_000\_000 | 47995: [2, 3, 5] ... [99998189999, 99998989999, 99999199999]  Test case 10 completed in 8.318 seconds. |  | S1:8.318  S2: |
| 11 | 1, 10\_000\_000\_000\_000 | 401696: [2, 3, 5] ... [9999970799999, 9999980899999, 9999987899999]  Test case 11 completed in 37.994 seconds. |  | S1:37.994  S2: |
| 12 | 1, 100\_000\_000\_000\_000 | 401696: [2, 3, 5] ... [9999970799999, 9999980899999, 9999987899999]  Test case 12 completed in 116.521 seconds. |  | S1:116.521  S2: |

### **Time complexities and big-O notations**

**Student 1’s (Raut, Nihaar) Analysis:**

The time complexity of the optimised code primarily relies on 2 main operations: palindrome generation and Miller-Rabin probabilistic primality testing.

For palindrome generation:

The palindrome\_machine function generates palindromic numbers within a specified range. It loops through different lengths of numbers, which ranges from the square root of the start number to the square root of the end number. Within each length, it iterates through digits to generate palindromes. As the range increases, the number of palindromes generated increases approximately at an under-root rate.

Therefore, the time complexity of palindrome generation can be estimated as **O(√(n)),** where n is the size of the range from start to end.

For Miller-Rabin primality testing:

The prime\_checker function incorporates the Miller-Rabin probabilistic primality test to check if a number is prime. It involves a series of modular exponentiation operations and iterations based on the number of witnesses (k).

The time complexity of the Miller-Rabin test is fixed as discussed previously which is

**O(k \* log^3(n)),** where k is the number of iterations and n is the number being tested.

Since k is typically set to a small constant (default is 3 in the code), the time complexity primarily depends on the magnitude of the number being tested.

Therefore, the overall time complexity of the primality testing can be approximated as **O(log^3(n)),** where n is the number being tested.

Overall Time Complexity:

Considering both operations, the overall time complexity of the provided code can be estimated as **O(√(n) + log^3(n)).** However, since the palindrome generation typically dominates the execution time for large ranges, the code can be considered to have a time complexity of **O(√ (n)).** This complexity indicates that the code's execution time grows at a slower rate than linearly with the size of the input range, making it efficient for practical use high range test cases.

Comparison with previously proposed original solution:  
  
**Improved Time Complexity:**

The Miller-Rabin primality testing algorithm has a fixed time complexity of **O(k \* log^3(n)),** where k is the number of iterations and n is the number being tested. However, for practical purposes, k is usually set to a small constant, making the time complexity ultimately dependent on the magnitude of the difference of the ranges of number being tested. This results in a more efficient prime checking process, especially for high order and large numbers.

Additionally, the palindrome generation function has a time complexity of **O(√ (n)),** where n is the size of the range from start to end. This complexity grows at a notably slower rate compared to the segmented sieve of Eratosthenes, which has a fixed time complexity of **O(n \* log(log(n))).**

**Space Efficiency:**

The Miller-Rabin primality testing algorithm does not require storing a list of primes, unlike the segmented sieve of Eratosthenes, which requires maintaining a sieve array to mark prime and composite numbers. This results in better space efficiency, especially for larger ranges where memory consumption can be significant.

**Student 2’s (Mohammed, Shuhaib) Analysis:**

### **Running time graphs**

# Reflecting on teamwork!

(Your task is to write about the limitations you faced while working together and give each team member a contribution mark that everyone agrees on.

Keep a weekly journal, noting down communication logs and the earned credits for each member. Summarise what each member did during the project.

There is a group mark decided by the marker. Each team member's overall contribution is assessed on a scale from 0% to 100%, with agreement from the team. For instance, if a member did not contribute to problem optimisation, they might receive 80% out of the 100%. An 80% individual effort could result in 80% of the group mark, but the final decision rests with the marker.

)

### **Contribution mark**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Name | ID | Task 1 (30%) | Task 2 (20%) | Task 3 (20%) | Task 4 (15%) | Task 5 (15%) | **Contribution mark**  **(100%)** |
| Last, First (Group leader) | 001234561 | 30% | 20% | 20% | 15% | 15% | 100% |
| Last, First (Champion 1) | 001234562 | 30% | 20% | 20% | 15% | 15% | 100% |
| Last, First (Champion 2) | 001234563 | 30% | 20% | 20% | 15% | 15% | 100% |
| Last, First (Role) | 001234564 | 20% | 20% | 10% | 0% | 10% | 60% |
| Last, First (Role) | 001234565 | 20% | 20% | 0% | 0% | 10% | 50% |
| Last, First (Role) | 001234566 | 0% | 0% | 0% | 0% | 0% | 0% |

### **Limitation discussion**

Your group might discuss the technical challenges, participation/engagement, collaboration, leadership, problem-solving skills, creativity and innovation, or communication dynamic topics.

### **Weekly journal**

|  |  |  |
| --- | --- | --- |
|  | **Task note** | **Status** |
| **Week 1: from date-date** |  |  |
| Last, First (Group leader) |  |  |
| Last, First (Champion 1) |  |  |
| Last, First (Champion 2) |  |  |
| Last, First (Role) |  |  |
| Last, First (Role) |  |  |
| Last, First (Role) |  |  |
| **Week 2: from date-date** |  |  |
| Last, First (Group leader) |  |  |
| Last, First (Champion 1) |  |  |
| Last, First (Champion 2) |  |  |
| Last, First (Role) |  |  |
| Last, First (Role) |  |  |
| Last, First (Role) |  |  |
| … |  |  |

# Reference

Tuan Vuong, COMP1819ADS, (2022), GitHub repository, <https://github.com/vptuan/COMP1819ADS>

# Appendix A.1 - Proposed solution 1 - 6

You can try to use Pycharm or VSCode to paste Python code into Word document. Note that it is important to keep the Python code in good structure, and text format for readability.

1. """
2. This video has NO Sound
4. Spyder Editor: Spyder 4.2.1
6. This demo is for Lab 02 - Ex1 MinMax function
7. """
8. **import** time
10. **def** minmax(sequence):
11. min = max = sequence[0] # assuming no-empty
12. **for** val **in** sequence:
13. **if** (val > max):
14. max = val
15. **if** (val < min):
16. min = val
17. **return** (min,max)
19. #print(minmax([1,2,3,5]))

22. **def** measure\_time(input\_size):
23. sequence = [i **for** i **in** range(input\_size)] # input = a list [0,1,2,...]
24. #print(sequence)
25. start = time.time() # start timer
26. **print**(minmax(sequence)) # execute the function with the sequence
27. **print**("Input size=", input\_size, " Time taken=", time.time()-start)

30. # Now, we make input size larger, 2k, 10k,50k, 200k,1000k
32. k = 1000;
33. measure\_time(2\*k)
34. measure\_time(10\*k)
35. measure\_time(50\*k)
36. measure\_time(200\*k)
37. measure\_time(1000\*k)
39. # Now, we plot in Excel. The plot looks linear? This is O(n) because
40. # the for loop in line 12.

# Appendix B - Test cases for correctness

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Input** | **Output** | **Comments** |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |
| 4 |  |  |  |
| 5 |  |  |  |
| 6 |  |  |  |

# Appendix C - Evidence of team contribution

Communication logs